UMLtoNoSQL: Automatic Transformation of Conceptual Schema to NoSQL Databases

Fatma ABDELHEDI
CBI2 – TRIMANE
Paris, France
fatma.abdelhedi@irit.fr

Amal AIT BRAHIM
IRIT, Toulouse Capitole University
Toulouse, France
amal.ait-brahim@irit.fr

Faten ATIGUI
CEDRIC- CNAM
Paris, France
faten.atigui@cnam.fr

Gilles ZURFLUH
IRIT, Toulouse Capitole University
Toulouse, France
gilles.zurfluh@irit.fr

***Abstract*—Volume, Variety and Velocity are the three dimensions that have definitely changed the tools we need to store and process Big Data effectively, giving rise to NoSQL systems for faster data access, better scalability and higher flexibility. While NoSQL systems have proven their efficiency to handle Big Data, it is still an unsolved problem how the automatic storage of Big Data in NoSQL systems could be done. One solution for addressing this problem is to model Big Data, and then define mapping rules towards the physical level. This paper proposes an automatic MDA-based approach that translates conceptual models expressed using the Unified Modeling Language (UML) into NoSQL physical models. Our approach rely on an intermediate logical model compatible with column, document and graph oriented systems which allows to choose the system type that suits the best with business rules and technical constraints.**

Keywords-UML conceptual model; NoSQL; Big Data storage; MDA; Models Transformation

I. CONTEXT AND RESEARCH PROBLEM

Big data have received a great deal of attention in recent years. Not only the amount of data is on a completely different level than before, but also we have different type of data including factors such as format, structure, and sources. In addition, the speed at which these data must be collected and analyzed is increasing. This has definitely changed the tools we need to benefit from Big Data, giving rise to new kinds of data management tools. NoSQL systems are a widely accepted tool able to support larger volumes of data by providing faster data access, better scalability and higher flexibility [16].

The lack of a model when creating a database is a key feature in NoSQL systems. In a table, attributes names and types are specified as and when the row is entered [8]. Unlike relational systems, where the model must be defined when creating the table, the schema less appears in NoSQL systems. This property offers undeniable flexibility that facilitates the evolution of models in NoSQL systems; but, it concerns exclusively the physical level (implementation) of a database [2]. A conceptual model is still required to define how data are stored and related in the database [3]. It provides a high level of abstraction and a semantic knowledge element close to human logic, which guarantees efficient data management [1]. The Unified Modeling Language (UML) has gained much attention in this area [1].

While NoSQL systems have proven their efficiency to handle Big Data, it is still an unsolved problem how the automatic storage of Big Data in NoSQL systems could be done. In our view, it is important to have a precise and automatic approach that helps and assists developer in the Big Database implementation task within NoSQL systems. One solution for addressing this problem is to model Big Data, and then define mapping rules towards the physical level. As discussed in the related work (see section 5), there are only few solutions that focus on mapping UML conceptual models into NoSQL physical models.

To overcome this situation, we propose the UMLtoNoSQL approach that automatically translates conceptual models expressed using the Unified Modeling Language (UML) into several NoSQL physical models. This approach is based on the Model Driven Architecture (MDA) especially known as a framework for models automatic transformations and allows the developer to choose the system type (column, document or graph) that suits the best with business rules and technical constraints.

The rest of the paper is structured as follows: Section 2 motivates our work using a case study in the healthcare field. Section 3 introduces our approach; two transformations processes are presented in this section, the first one creates a NoSQL generic model starting from a UML conceptual model, and the second one generates NoSQL physical models from this generic model; Section 4 details our experiments; Section 5 reviews previous work on models transformation; Section 6 provides a discussion on our approach and announces future work. Finally, Section 7 concludes the paper.

II. ILLUSTRATIVE EXAMPLE

To motivate our work and illustrate the different steps of our approach, we introduce in this section an example of Big Data application in the healthcare filed. This application concerns international scientific programs for monitoring patients suffering from serious diseases. The main goal of this program is (1) to collect data about diseases development over time, (2) to study interactions between different diseases and (3) to evaluate the short and medium-term effects of their treatments. The medical program can last up to 3 years.

DOI 10.1109/AICCSA.2017.76

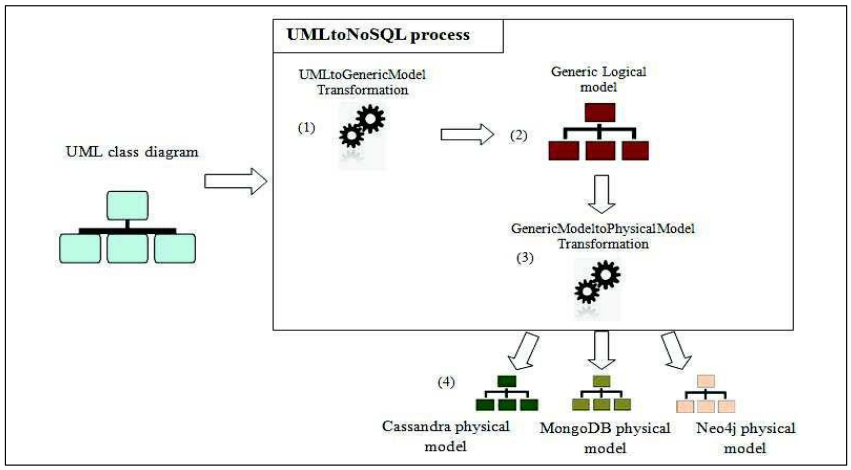


Figure 1. Overview of UMLtoNoSQL process.

Data collected from establishments involved in this kind of program have the features of Big Data (the 3 V):
Volume: the amount of data collected from all the establishments in three years can reach several terabytes.
Variety: data created while monitoring patients come in different types; it could be (1) structured as the patient's vital signs (respiratory rate, blood pressure, etc.), (2) semi-structured document such as the package leaflets of medicinal products, (3) unstructured such as consultation summaries, paper prescriptions and radiology reports.
Velocity: some data are produced in continuous way by sensors; it needs a [near] real time process because it could be integrated into a time-sensitive processes (for example, some measurements, like temperature, require an emergency medical treatment if they cross a given threshold).

This is a typical example in which the use of a NoSQL system is suitable. As mentioned. This kind of systems operate on schema less data model enabling users to quickly and easily incorporate new data into their applications without rewriting tables. Nevertheless, there is still a need for a semantic model to know how data are structured and related in the database; this is particularly necessary to write declarative queries where tables and columns names are specified.

UML is widely accepted as a standard modelling language for describing complex data. In the medical application, briefly presented above, the database contains structured data, data of various types and formats (explanatory texts, medical records, x-rays, etc.), and big tables (records of variables produced by sensors). Therefore, we choose the UML class diagram to describe the medical data.

III. CONTRIBUTION

Our purpose in this paper is to assist developers in storing Big Data in NoSQL systems. For this, we propose the UMLtoNoSQL approach that automatically transforms a UML conceptual model describing Big Data into a NoSQL physical model.

In our approach, we introduce a logical level between conceptual (business description) and physical (technical description) levels in which a generic logical model is developed. This logical model exhibits a sufficient degree of independence so as to enable its mapping to one or more NoSQL platforms. Developers will benefit from it in two ways: (1) it describes data according to the common features of NoSQL models (column, document and graph), which allow it's mapping on several platforms. (2) it abstracts out technical details of NoSQL systems, this mean that the logical level remains stable, even though the NoSQL system evolves over time. In this case, it would be enough to evolve the physical model, and of course adapt the transformation rules; this simplifies the transformation process and saves time for developers.

To formalize and automate UMLtoNoSQL process, we use the Model Driven Architecture (MDA) proposed by the OMG [4]. One of the main aims of MDA is to separate the functional specification of a system from the details of its implementation in a specific platform. This architecture defines a hierarchy of models from three points of view: Computation Independent Model (CIM), Platform Independent Model (PIM), and Platform Specific Model (PSM) [5]. Among this proposed models, we use:

UMLtoGenericModel (1) is the first transformation (section 3.1) in UMLtoNoSQL process. It is in charge of converting the input UML class diagram (conceptual PIM) into the generic logical model (2) conforming to the generic logical metamodel proposed in Section 3.1.2; this metamodel describes a data structure compatible with the three types of NoSQL systems. GenericModeltoPhysicalModel (3) is the second transformation (section 3.2) in UMLtoNoSQL process. It is in charge of transforming the generic logical model into NoSQL physical models (PSMs) (4).

We note that UMLtoNoSQL process generates several NoSQL physical models from a UML class diagram. In order to do this, it's necessary to register, for each physical model, its specific parameters (transformation rules). To illustrate our work, we have taken as example three physical models that correspond

to: Cassandra, MongoDB and Neo4j systems. If the developer chooses to use another system, the process must be completed by adding new parameters specific to this system.

A. UMLtoGenericModel Transformation

In this section we present the UMLtoGenericModel transformation, which is the initial step in our approach presented in Figure 1. We first define the source (UML Class Diagram) and the target (Generic Logical Model), and then we focus on the transformation itself.

Source: A Class Diagram (CD) is defined as a tuple (N, C, L), where:

N is the class diagram name,

C is a set of classes. Classes are composed from structural and behavioral constituents; in this paper, we consider only the structural part. Since the operations are linked to the behavior, we will not take them into account. The schema of each class c ∈ C is a tuple (N, A, Ident0^c), where:

- c.N is the class name.
- c.A = $\{a_1^c, \dots, a_q^c\}$ is a set of q attributes. The schema of each attribute $a^c \in$ A is a pair (N,C) where "a^c.N" is the attribute name and "a^c.C" the attribute type; C can be a predefined class, i.e. a standard data type (String, Integer, Date ...) or a business class (class defined by user).
- c. Ident0^c is a special attribute of c; it has a name Ident0^c.N and a type called "Oid". In this paper, an attribute whose type is "Oid" represents a unique object identifier, i.e. an attribute whose value distinguishes an object from all other objects of the same class.

L is a set of links. Each link l between n classes, with n>=2, is defined as a tuple (N, Ty, Pr^l), where:

- l.N is the link name.
- l.Ty is the link type. In this paper, we will only consider the three main types of links between classes: Association, Composition and Generalization.
- $\text{l.Pr}^l = \{\text{pr}_1^l, \dots, \text{pr}_n^l\}$ is a set of n pairs. ∀ i ∈ {1,..,n}, $\text{pr}_i^l = (c, \text{cr}^c)$, where pr_i^l.c is a linked class and pr_i^l. cr^c is the cardinality placed next to c. Note that pr_i^l. cr^c can contain a null value if no cardinality is indicated next to c (like in generalization link).

Class diagram metamodel is shown in Figure 2; this metamodel is adapted from the one proposed by the OMG [7].

Target: The target of UMLtoGenericModel transformation corresponds to a generic logical model that describes data according to the common features of the three types of NoSQL systems: column-oriented, document-oriented and graph-oriented. In the generic logical model, a DataBase (DB) is defined as a tuple (N, T, R), where:

N is the database name,

T is a set of tables. The schema of each table t ∈ T is a tuple (N, A, IdentL^t), where:

- t.N is the table name.
- $\text{t.A} = \{\text{a}_1^t, \dots, \text{a}_q^t\}$ is a set of q attributes that will be used to define rows of t; each row can have a variable number of attributes. The schema of each attribute $\text{a}^t \in$ A is a pair (N,Ty) where "a^t.N" is the attribute name and "a^t.Ty" the attribute type.
- t. IdentL^t is a special attribute of t; it has a name IdentL^t.N and a type called "Rid". In this paper, an attribute whose type is "Rid" represents a unique row identifier, i.e. an attribute whose value distinguishes a row from all other rows of the same table.

R is a set of binary relationships. In the generic logical model there are only binary relationships between tables. Each relationship r ∈ R between t_1 and t_2 is defined as a tuple (N, Pr^r), where:

- r.N is the relationship name.
- $\text{r.Pr}^r = \{\text{pr}_1^r, \text{pr}_2^r\}$ is a set of two pairs. ∀ i ∈ {1,2}, $\text{pr}_i^r = (\text{t}, \text{cr}^t)$, where pr_i^r.t is a related table and pr_i^r. cr^t is the cardinality placed next to t.

Metamodel of the proposed generic logical model is shown in Figure 3. Note that the attribute value may be either atomic or complex (set of attributes). We represent this by using the XOR constraint (UML predefined constraint).

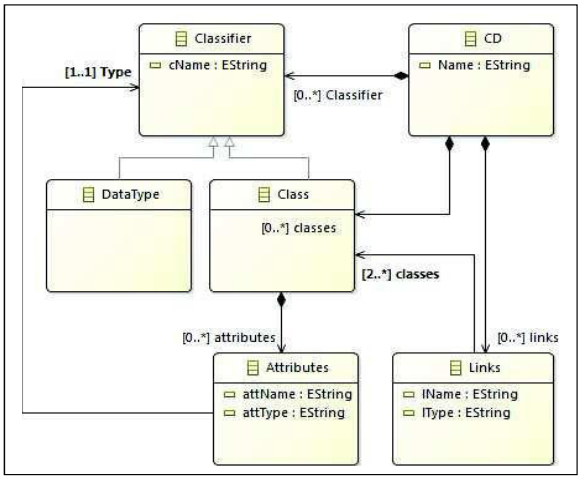


Figure 2. Source Metamodel.

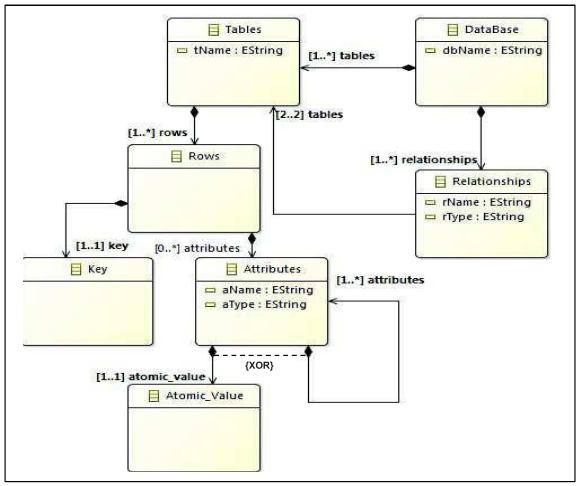


Figure 3. Target Metamodel.

Transformation Rules:

R1: each class diagram CD is transformed into a database DB, where DB.N = CD.N.

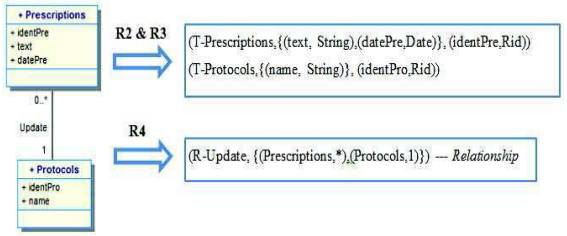
R2: each class c ∈ C is transformed into a table t ∈ DB, where t.N = c.N, $IdentL^t$.N = $IdentO^c$.N.

R3: each attribute a^c ∈ c.A is transformed into an attribute a^t, where a^t.N = a^c.N, a^t.Ty = a^c.C, and added to the attribute list of its transformed container t such as a^t ∈ t.A.

R4: each binary link l ∈ L (regardless of its type: Association, Composition or Generalization) between the classes c_1 and c_2 is transformed into a relationship r ∈ R between the tables t_1 and t_2 representing c_1 and c_2, where r.N = l.N, r.Pr^r = {(t_1, cr^{c_1}),(t_2, cr^{c_2})}, cr^{c_1} and cr^{c_2} are the cardinality placed respectively next to c_1 and c_2.

Example:

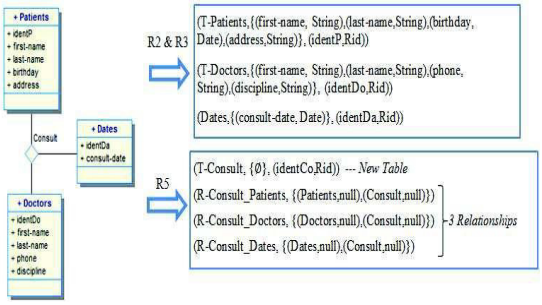
As an example, consider the classes "Prescriptions" and "Protocols", and the link "Update". Applying the transformation rules R2 and R3 generates the tables "T-Prescriptions" and "T-Protocols". As soon as R4 applies, the appropriate transformation is made, and the link "Update" is transformed into a relationship "R-Update" as follow:



R5: each link l ∈ L between n classes {c_1, … , c_n} (n>=3) is transformed into (1) a new table t^l, where t^l.N = l.N and t^l.A = ∅, and (2) n relationships {r_1, … , r_n}, ∀ i ∈ {1,..,n} r_i links t^l to another table t_i representing a related class c_i, where r_i.N = (t^l.N)_(t_i.N) and r_i. Pr^r = {(t^l, null), (t_i, null)}.

Example:

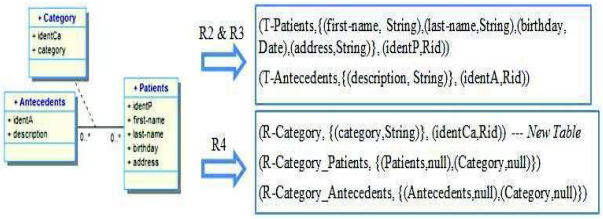
In the example shown below, the n-ary link "Consult" that relates the classes: "Patients", "Doctors" and "Dates" is transformed by applying R5 into: (1) a new table "T-Consult" that does not contain attributes and (2) three relationships: "R-Consult_Patients", "R-Consult_Doctors" and "R-Consult_Dates" that relate "T-Consult" respectively to "T-Patients", "T-Doctors" and "T-Dates".We note that the tables "T-Patients", "T-Doctors" and "T-Dates" were created by applying R2 and R3.



R6: each association class c_{asso} between n classes {c_1, … , c_n} (n>=2) is transformed like a link between multiple classes (R5) using (1) a new table t^{ac}, where t^{ac}.N = l.N, and (2) n relationships {r_1, … , r_n}, ∀ i ∈ {1,..,n} r_i links t^{ac} to another table t_i representing a related class c_i, where r_i.N = (t^{ac}.N)_(t_i.N) and r_i. Pr^r = {(t^{ac}, null), (t_i, null)}. Like any other table, t^{ac} contain also a set of attributes A, where t^{ac}.A = c_{asso}.A.

Example:

The association class "Category" that links the classes: "Patients" and "Antecedents" is transformed by applying R6 into: (1) a new table "T-Category" that contain the attribute "category" and (2) two relationships: "R-Category_Patients" and "R-Category_Antecedents" that relate "T-Category" respectively to "T-Patients" and "T-Antecedents".



These transformation rules have also been specified with QVT (Query / View / Transformation), which is a standard defined by OMG for expressing models transformation. An excerpt from QVT rules is shown in Figure 6.

B. GenericModeltoPhysicalModel Transformation

In this section we present the GenericModeltoPhysicalModel transformation, which is the second step in our approach UMLtoNoSQL (figure 1). It is in charge of creating NoSQL physical models from the proposed generic logical model.

Source: The source of GenericModeltoPhysicalModel transformation is the target of the previous UMLtoGenericModel transformation.

Target: To illustrate our approach, we have chosen: Cassandra (column-oriented), MongoDB (documents-oriented) and Neo4j (graph-oriented); three well known NoSQL systems.

In Cassandra physical model, KeySpace (KS) is the top-level container that owns all the elements. It's defined as a tuple (N, F), where:

N is the keyspace name,

F is a set of columns-families. The schema of each columns family f ∈ F is a tuple (N, Cl, $PrimaryKey^f$), where:

- f.N is the columns-family name,
- f.Cl = {cl_1, … , cl_q} is a set of q columns that will be used to define rows of f; each row can have a variable number of columns. The schema of each column cl ∈ Cl is a pair (N,Ty) where "cl.N" is the column name and "cl.Ty" the column type.
- f. $PrimaryKey^f$ is a special column of f; it has a name $PrimaryKey^f$.N and a type $PrimaryKey^f$.Ty (standard data type). $PrimaryKey^f$ identifies uniquely each row of f.

In MongoDB physical model, DataBase (DB^{MD}) is the top-level container that owns all the elements. It's defined as a tuple (N, Cll), where:
N is the database name,
Cll is a set of collections. The schema of each collection cll ∈ Cll is a tuple (N, Fl,Id^{cll}), where:

- cll.N is the collection name,
- cll.Fl = $Fl^{A} \cup Fl^{CX}$ is a set of atomic and complex fields that will be used to define rows, called documents, of Cll. Each document can have a variable number of fields. The schema of an atomic field $fl^{a} \in Fl^{A}$ is a tuple (N,Ty) where "fl^{a}.N" is the field name and "fl^{a}.Ty" is the field type. The schema of a complex field $fl^{cx} \in Fl^{CX}$ is also a tuple (N, Fl') where fl^{cx}.N is the field name and fl^{cx}.Fl' is a set of fields where Fl'⊂ Fl.
- cll. Id^{cll} is a special field of cll; it has a name Id^{cll}.N and a type Id^{cll}.Ty (standard data type). Id^{cll} identifies uniquely each document of cll.

In Neo4j physical model, Graph (GR) is the top-level container that owns all the elements. It's defined as a tuple (V, E), where:
V is a set of vertex. The schema of each vertex v ∈ V is a tuple (L, Pro, Id^{v}), where:

- v.L is the vertex label,
- v.Pro = $\{pro_1, \dots, pro_q\}$ is a set of q properties. The schema of each property pro ∈ Pro is a pair (N,Ty), where "pro.N" is the property name and "pro.Ty" the property type.
- v. Id^{v} is a special property of v; it has a name Id^{v}.N, a type Id^{v}.Ty and the constraint "Is Unique ". It identifies uniquely v in the graph.

E is a set of edges. The schema of each edge e ∈ E is a tuple (L, v_1, v_2), where:

- e.L is the edge label,
- e. v_1 and e. v_2 are the vertexes related by e.

Transformation Rules: For some NoSQL systems, many solutions can ensure the implementation of the generic logical model. In order to choose the most suitable solution, the developer can use the performance measurement shown in Section 4.2. These measurements concern the response time of queries that accesses to two related tables; the relationship between these tables has been implemented according to the different solutions shown below. The developer will make his choice according to the characteristics of queries he wants to perform and the expected performances.

We note that the set of solutions proposed in this section is not inclusive; more marginal solutions may be considered.

To Cassandra physical model:
R1: each database DB is transformed into a keyspace KS, where KS.N = DB.N.
R2: each table t ∈ DB is transformed into a columns-family f ∈ KS, where f.N = t.N, $PrimaryKey^{f}$.N = $IdentL^{t}$.N.
R3: each attribute a^{t} ∈ t.A is transformed into a column cl, where cl.N = a^{t}.N, cl.Ty = a^{t}.Ty, and added to the column list of its transformed container f such as cl ∈ f.Cl.
R4: As Cassandra does not support imbrication; the only solution we can use to express relationships between columns-families consists in using reference columns. A reference column is a monovalued or multivalued column in one columns-family whose values must have matching values in the primary key of another columns-family; we note that this constraint is not automatically managed by the system Cassandra; it remains the responsibility of the user to check it.

For each relationship r between two tables t_1 and t_2, three solutions could be considered:
Solution 1: r is transformed into a reference column cl referencing f_2 (the columns-family representing t_2), where cl.N = (f_2.N)_Ref and cl.Ty = $PrimaryKey^{f2}$.Ty, and then added to the columns list of f_1 (the columns-family representing t_1) such as cl ∈ f_1.Cl. While instantiating f_1, the value of the reference column cl will correspond to one or many values in the primary key of f_2.
Solution 2: r is transformed into a reference column cl referencing f_1 (the columns-family representing t_1), where cl.N = (f_1.N)_Ref et cl.Ty = $PrimaryKey^{f1}$.Ty, and then added to the columns list of f_2 (the columns-family representing t_2) such as cl ∈ f_2.Cl. While instantiating f_2, the value of the reference column cl will correspond to one or many values in the primary key of f_1.
Solution 3: r is transformed into a new columns-family f composed of two reference columns referencing the columns-families f_1 and f_2 representing the related tables t_1 and t_2, where f.N = r.N, f.Cl = $\{cl_1, cl_2\}$, cl_1.N = (f_1.N)_Ref, cl_1.Ty = $PrimaryKey^{f1}$.Ty, cl_2.N = (f_2.N)_Ref and cl_2.Ty = $PrimaryKey^{f2}$.Ty.

A reference column can either be monovalued or multivalued. Table 1 indicates the type of the reference column according to the relationship cardinalities and the transformation solution used.

TABLE I. DESCRIPTIVE TABLE OF REFERENCE COLUMN TYPES

Relationship	Solution	Reference column type
r = (N,{(t_1,*),(t_2,1)})	Solution 1	Monovalued
	Solution 2	Multivalued
	Solution 3	Monovalued
r = (N,{(t_1,1),(t_2,1)})	Solution 1	Monovalued
	Solution 2	Monovalued
	Solution 3	Monovalued
r = (N,{(t_1,*),(t_2,*)})	Solution 1	Multivalued
	Solution 2	Multivalued
	Solution 3	Monovalued

To MongoDB physical model:
R1: each database DB is transformed into a MongoDB database DB^{MD}, where DB^{MD}.N = DB.N.
R2: each table t ∈ DB is transformed into a collection cll ∈ DB^{MD}, where cll.N = t.N et Id^{cll}.N = $IdentL^{t}$.N.
R3: each attribute a^{t} ∈ t.A is transformed into an atomic field fl^{a}, where fl^{a}.N = a^{t}.N, fl^{a}.Ty = a^{t}.Ty, and added to

the field list of its transformed container cll such as fl ∈ cll. Fl^A.

R4: relationships in MongoDB could be transformed by using reference fields or embedding. A reference field is a monovalued or multivalued field in one collection whose values must have matching values in the Id of another collection; checking this constraint remains the responsibility of the user.

For each relationship r between two tables t_1 and t_2, five solutions could be considered:

Solution 1: r is transformed into a reference field fl referencing cll_2 (the collection representing t_2), where fl.N = (cll_2.N)_Ref and fl.Ty = Id^{cll_2}.Ty, and then added to the fields list of cll_1 (the collection representing t_1) such as fl ∈ cll_1. Fl^A.

Solution 2: r is transformed into a reference field fl referencing cll_1 (the collection representing t_1), where fl.N = (cll_1.N)_Ref and fl.Ty = Id^{cll_1}.Ty, and added to the field list of cll_2 (the collection representing t_2) such as fl ∈ cll_2. Fl^A.

Solution 3: r is transformed by embedding the collection cll_2 representing t_2 in the collection cll_1 representing t_1, where cll_2 ∈ cll_1. Fl^{CX}.

Solution 4: r is transformed by embedding the collection cll_1 representing t_1 in the collection cll_2 representing t_2, where cll_1 ∈ cll_2. Fl^{CX}.

Solution 5: r is transformed into a new collection cll, where cll.N = r.N, cll.Fl = {fl_1, fl_2}, fl_1.N = (cll_1.N)_Ref, fl_1.Ty = Id^{cll_2}.Ty, fl_2.N = (cll_2.N)_Ref and fl_2.Ty = Id^{cll_2}.Ty, where cll_1 and cll_2 are the collections representing t_1 and t_2.

Each reference field used in Solution 1, 2 and 5 can either be monovalued or multivalued. Table 2 indicates the type of the reference field according to the relationship cardinalities and the transformation solution used.

TABLE II. DESCRIPTIVE TABLE OF REFERENCE FIELD TYPES

Relationship	Solution	Reference field type
r = (N,{(t_1,*),(t_2,1)})	Solution 1	Monovalued
	Solution 2	Multivalued
	Solution 5	Monovalued
r = (N,{(t_1,1),(t_2,1)})	Solution 1	Monovalued
	Solution 2	Monovalued
	Solution 5	Monovalued
r = (N,{(t_1,*),(t_2,*)})	Solution 1	Multivalued
	Solution 2	Multivalued
	Solution 5	Monovalued

To Neo4j physical model:

R1: each table t ∈ DB is transformed into a vertex v ∈ V, where v.L = t.N, Id^v.N = $IdentL^t$.N.

R2: each attribute a^t ∈ t.A is transformed into a property pro, where pro.N = a^t.N, pro.Ty = a^t.Ty, and added to the property list of its transformed container v such as pro ∈ v.Pro.

R3: Each relationship r between two tables t_1 and t_2 is transformed into an edge e, where e.L = r.N, relating two vertex v_1 and v_2, where v_1 and v_2 are the vertex representing t_1 and t_2.

IV. EXPERIMENTS

In this section we show how to implement the UMLtoNoSQL process according to each transformation solution proposed in section 3.2. Then we evaluate the performances of each solution in order to assist developer in choosing the most effective one.

A. Implementation

We have implemented UMLtoGenericModel and GenericModeltoPhysicalModel transformations using a set of tools provided by Eclipse Modeling Framework (EMF). Each transformation is expressed as a sequence of elementary steps that builds the resulting model step by step from the source model. First, we create the source and the target metamodels. Second, we build an instance of the source metamodel; for this, we use the standard-based XML Metadata Interchange (XMI) format. Third, we implement the transformation rules by means of the Query / View / Transformation (QVT) language (the OMG standard language for specifying model transformations). Finnaly, we test the transformation by running the QVT script; the result is provided in the form of XMI file. Note that due to lack of space, we only present excerpts from models and QVT scripts.

UMLtoGenericModel is the first transformation in UMLtoNoSQL process. It transforms the input UML class diagram (figure 4) into the proposed generic logical model (figure 5). This model is conform to the generic logical metamodel (figure 3) presented in Section 3.1. This transformation is performed by means of the transformation rules defined in section 3.1.3. An excerpt from the QVT script is shown in Figure 6; the comments in the script indicate the rules used.

GenericModeltoPhysicalModel is the second transformation in UMLtoNoSQL process. It takes as input the generic logical model generated by the previous transformation (UMLtoGenericModel) and return as output a NoSQL physical model. As presented in section 3.2, the generic logical model proposed in this paper does not imply a specific system; several instances can be generated from this model to target a specific NoSQL system (Cassandra, MongoDB or Neo4j). Depending on the target system functionalities, relationships of the logical model could be converted into different forms. We have proposed different solutions to transform these relationships under Cassandra and MongoDB (3 solutions for Cassandra and 5 for MongoDB). Lacks of place, we only present two implementations of the generic logical model. The first one was performed on Cassandra according to solution 1 (figure 7) and the second one was done within MongoDB according to solution 4 (figure 8).

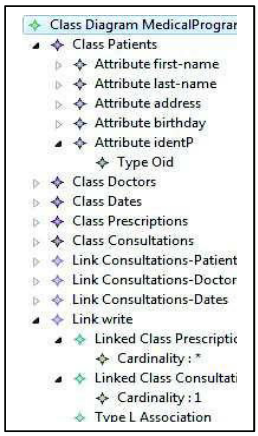


Figure 4. Source Model.

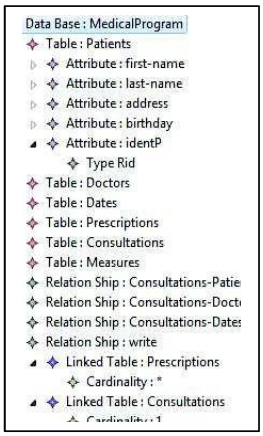


Figure 5. Target Model.

```
modeltype UML uses "http://UMLClassDiagram.com";
modeltype COLM uses "http://GenericLogicalModel.com";
transformation TransformationUmlToColumnsOrientedModel
(in Source: UML, out Target: COLM);main()
{Source.rootObjects()[ClassDiagram]
 -> map toDataBase();}
-- Transforming Class Diagram to DataBase
mapping ClassDiagram::toDataBase():DataBase{name := self.name;
table:=self.classes -> map toClass();relationship:=self.links -> toRelationship();}
-- Transforming Class to Table
mapping UML ::Class::toClass():COLM::Table{name:=self.name;attributet:=
self.attributec -> map toAttribute();}
-- Transforming Attribute to Column
mapping UML ::Attribute::toAttribute():COLM::Attribute{name:=self.name;
typea:=self.typea -> map toType(); }
mapping UML ::Type::toType():COLM::Type{typea:=self.typea;}
mapping UML ::Link::toRelationship():COLM::RelationShip{name:=self.name;
```

Figure 6. UMLtoGenericModel Transformation

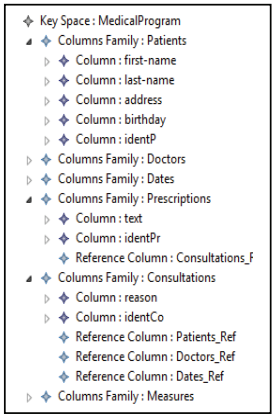


Figure 7. CassandraModel.

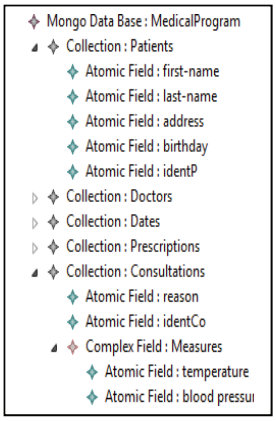


Figure 8. MongoDBModel.

B. *Evaluation*

Once the NoSQL physical model (Cassandra or MongoDB model) has been created according to each proposed solution, an evaluation has to be performed to study the impact that the choice of the solution used may have on the execution time of queries. The graph-oriented system Neo4j does not offer many solutions to implement relationships; therefore, the developer does not need to choose between several solutions.

We carry out the experimental assessment using a cluster made up of 3 machines. Each machine has the following specifications: Intel Core i5, 8GB RAM and 2TB disk. On the other hand, we have used data generator tools to generate a dataset of about 1TB with CSV format for Cassandra and JSON format for MongoDB; these files are loaded into the systems using shell commands.

We have written 6 queries; each query concerns two tables and the relationship between them. The complexity of these queries increases gradually; the simplest one applies a filter to a table and returns attributes of the other table; the most complex applies several filters and returns attributes of the two related tables. We note that the concepts "table" and "attribute" correspond respectively to "columns-family" and "column" in Cassandra or "collection" and "field" in MongoDB.

An excerpt from our experiment results is depicted in Figure 9. For each query, we indicate the obtained response time according to (1) the relationship cardinalities and (2) the used transformation solution.

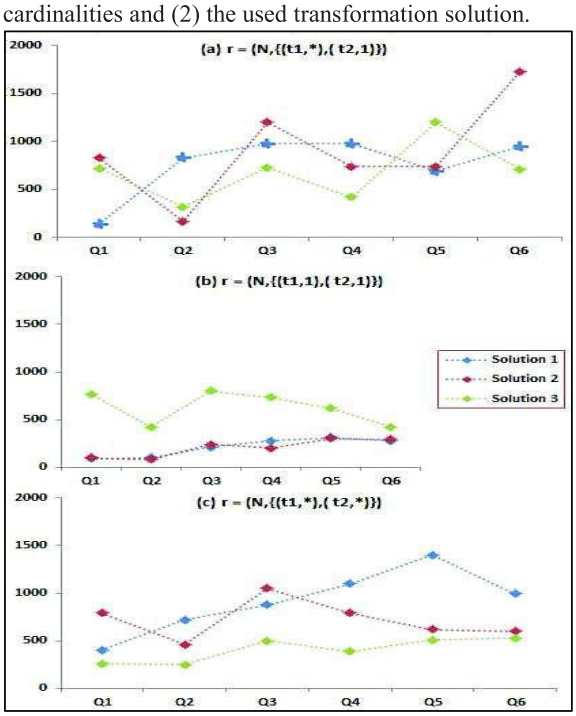


Figure 9. Experimental Results.

To implement a relationship under Cassandra or MongoDB, the developer can use our experiment results and choose the most suited solution according to the following criteria: (1) The features of each query that uses this relationship (number of filters, number of attributes to return), (2) The time response and (3) How frequently this query is used.

V. RELATED WORK

NoSQL systems have proven their efficiency in terms of flexibility and handling Big Data. In the literature, a number of researchers have proposed approaches for transforming the multidimensional conceptual model to a NoSQL model. For example, Dehdouh et al. [12] propose three approaches to map a multidimensional model into a logical model adapted to column-oriented NoSQL systems. While this approach has the advantage to start from the conceptual level, the proposed models are Domain-Specific (Data Warehouses system). So they

consider fact, dimension, and typically one type of links only.

Other studies [10] and [9] have investigated the process of transforming relational databases into a NoSQL model. Li [10] have proposed an approach for transforming a relational database into HBase (column-oriented system). Vajk et al. [9] defined a mapping from a relational model to document-oriented model using MongoDB. These works rely on the relational model that, unlike UML class diagram, lacks of semantic richness, especially through the several types of relationships that exist between classes.

To the best of our knowledge, only few works have presented approaches to implement UML conceptual model into NoSQL systems. Li et al. [11] propose a MDA-based process to transform UML class diagram into column-oriented model specific to HBase. Starting from the UML class diagram and HBase metamodels, authors have proposed mapping rules between the conceptual level and the physical one. Obviously, these rules are applicable to HBase, only. Gwendal et al. [3] describe the mapping between a UML conceptual model and graph databases via an intermediate graph metamodel. In this work, the transformation rules are specific to graph databases used as a framework for managing complex data with many connections. Generally, this kind of NoSQL systems is used in social networks where data are highly connected. Reagrding this state of the art, [11] and [3] consider, each, a single type of NoSQL systems (column-oriented in [11] and graph-oriented in [3]). However, it makes more sense to choose the target system according to the user's needs. For example, if processing operations requires access to hierarchically structured data, the document-oriented system proves to be the most adapted solution.

The main purpose of our work is to assist developers in storing Big Data in NoSQL systems. For this, we have proposed a new MDA-based approach that transforms a UML conceptual model describing Big Data into several NoSQL physical models. This automatic process allows the developer to choose the system type (column, document or graph) that suits the best with business rules and technical constraints.

VI. Discussion & Future Work

Big Data applications developers have to deal with the question: how to store Big Data in NoSQL systems? To address this problem, we have proposed the UMLtoNoSQL approach, a MDA-based approach, to implement UML conceptual models describing Big Data in NoSQL systems. Our approach rely on a pivotal model (the "Generic Logical Model" in the paper) designed for NoSQL systems. Instances of this model can be generated to target specific NoSQL system (Cassandra, MongoDB, Neo4j, etc.).

In this approach, we have considered some constraints such as data type and identifier uniqueness. However, other constraints are required as context restrictions to further refine the semantics of the UML conceptual model elements. These constraints are defined by well-formedness rules expressed using the Object Constraint Language (OCL). It's a widely accepted standard allowing the specification of formal constraints on conceptual models in a declarative way similar to predicate logic [13].

Once the NoSQL physical model is created, another process has to be performed to check the OCL constraints defined in the conceptual model. Checking OCL constraints at the physical level is a real challenge within NoSQL systems since the vast majority of NoSQL approaches lack any advanced mechanism for integrity constraint checking. Considering this limitation, we plan to complete and generalize the UMLtoNoSQL approach by taking into account other more complex constraint.

VII. Conclusion

This paper is about an investigation on the implementation of Big Data base within NoSQL systems. We have proposed respectively, two transformations for this purpose, UMLtoGenericModel and GenericModeltoPhysicalModel, in order to generate a physical model from a UML class diagram. Using MDA formalism, we define rules to transform the conceptual model to NoSQL systems automatically. Our contribution illustrates an intermediate unified logical model between UML conceptual model and NoSQL physical model. Our logical model is compatible with column, document and graph oriented systems. This model uses tables and binary relationships that link them. The independence between the three physical models is ensured. Furthermore, we propose different solutions to transform the binary relationships of the logical model under Cassandra and MongoDB. Depending on the system functionalities, the binary relationships could be converted into different forms.

Our experimental work demonstrates an evaluation of NoSQL physical model, Cassandra or MongoDB model, according to each proposed solution. The developer can then use our experimental results and choose the most suited solution according to the criteria considered in the paper.

References

[1] A. Abello, "Big Data Design". In DOLAP, 2015.

[2] Herrero, V., Abelló, A., & Romero, O. NOSQL Design for Analytical Workloads: Variability Matters. In ER, 2016.

[3] D. Gwendal, S. Gerson, C. Jordi. "UMLtoGraphDB: Mapping Conceptual Schemas to Graph Databases". In ER, 2016.

[4] J. Hutchinson, M.Rouncefield, and J.Whittle. "Model-driven engineering practices in industry". In ICSE, 2011.

[5] J. .Bézivin and O. Gerbé. "Towards a Precise Definition of the OMG/MDA Framework". In ASE, 2001.

[6] D.Abadi , S.Madden , N.Hachem, "Column-stores vs. row-stores". In international conference on Management of data, 2008.

[7] http://www.omg.org/spec/UML/2.5/

[8] A. Angadi, Ak. Angadi, Karuna. Gull. "Growth of New Databases & Analysis of NOSQL Datastores". In IJARCSSE. 2013.

[9] T. Vajk, P. Feher, K. Fekete, H. Charaf. "Denormalizing data into schema-free databases". In CogInfoCom, 2013.

[10] C. Li. "Transforming relational database into HBase: A case study". In ICSESS, 2010.

[11] Yan Li, Ping Gu. "Transforming UML Class Diagrams into HBase Based on Meta-model. Information Science". In ISEEE, 2014

[12] Dehdouh, K., Bentayeb, F., Boussaid, O., Kabachi, N. Using the column oriented model for implementing big data warehouses. In PDPTA, 2015.

[13] http://www.omg.org/spec/OCL/2.4/